WebGL
Agenda

• Rendering pipeline
• Boilerplate for minimal application
  • Obtaining rendering context
  • Uploading data to GPU
• Transformations
• Shaders
• Textures
A bit of background!

- WebGL is a low-level, rendering API for use within browsers.
- Provides access to the GPU
- Requires quite a bit of code overhead
- Current version WebGL is 1.0 (specification for 2.0 is ready)
- Based on OpenGL ES 2.0
- Programmable pipeline!

THREE.js examples

- [http://threejs.org/examples/](http://threejs.org/examples/)
Rendering pipeline

Buffers (positions, normals, etc...) → Vertex Shader → Fragment Shader

Uniforms → Vertex Shader

Attributes → Fragment Shader

gl_Position → Fragment Shader

gl_FragColor

Image credit: Gregg Tavares
Rasterization

- WebGL, OpenGL, DirectX are what we call rasterizers
- We specify data in ‘continuous’ space that gets rasterized (digitized)
A Simple Program

• Any WebGL program will have similar structure
  • Create context
  • Upload and compile shaders
  • Upload drawing data into buffers
  • Render!

• All these processes share similar syntax of `gl.createX()`, `gl.bindX()` …
A Simple Program
Rendering Context Creation

- Rendering context can be obtained through HTML `<canvas>` element
webgl-utils.js

- Small useful utility from google
- Context creation
- Animation (later)
  - Single line setup!

```javascript
gl = WebGLUtils.setupWebGL(canvas);
```

- Download
Shaders

• Basically what the graphics programming is all about

• WebGL has two types of shaders
  • Vertex Shaders
  • Fragment Shaders

• Can be specified within the HTML a `<script>` tag, as well as outside in their own files

• Written in WebGL Shading Language

• Compiled and linked as your C program
WebGL Shading Language

• Similar to C
  • Standard flow control
  • Some additional data structures
    • vec2, vec3, vec4, mat3, mat4
  • Standard operators should work on these types
  • Component-wise matrix multiplication `matrixCompMult(mat x, mat y)`
  • Vector comparison functions - `greaterThan(T x, T y)`,
  • Geometrical functions (`dot(T x, T y)`, `cross(vec3 x, vec3 y)`, etc.)
  • Swizzling
    • `vec4 v1(1.0, 2.0, 1.0, 0.0); vec2 v2 = v1.zz;`
• Very good summation of the language features: WebGL Reference Card
Vertex Shader

- Small program run per vertex of your input geometry
- JavaScript application will upload data to Vertex Shader attributes
- Attribute is data that you store per vertex
  - position, color, normal, etc.
- outputs special `gl_Position` variable

```glsl
attribute vec3 aVertexPosition;
attribute vec3 aVertexColor;

uniform mat4 uModelView;
uniform mat4 uProjection;

varying vec3 vVertexColor;

void main(void) {
    gl_Position = uProjection * uModelView * vec4(aVertexPosition, 1.0);
    vVertexColor = aVertexColor; // passthrough
}
```
Fragment Shader

• Small program run per each fragment
• Most of the magic happens here
  • ShaderToy - all fragment shaders!
• Outputs the \texttt{gl\_FragColor}, which might become the color of your pixel

```glsl
precision mediump float; // required by webGL
varying vec3 vVertexColor;

void main(void) {
  gl_FragColor = vec4( vVertexColor, 1.0);
}
```
Shading Language variable qualifiers

• attribute
  • Linkage between a vertex shader and per-vertex data

• uniform
  • Value does not change across the primitive being processed, constant for all the vertices.

• varying
  • Link between the vertex shader and the fragment shader for interpolated data
Access Point to Shader variables

• How to get access to the input variables in shaders?
  • Required when drawing!
• For attributes we need to
  • Query attribute location (by name specified in the shader)
  • Tell WebGL that we intending on using it
• For uniforms
  • Query attribute location (by name specified in the shader)

```javascript
shaderProgram.vertexPosAttrib = gl.getAttribLocation(shaderProgram, "aVertexPosition");
gl.enableVertexAttribArray( shaderProgram.vertexPosAttrib);

shaderProgram.vertexColAttrib = gl.getAttribLocation(shaderProgram, "aVertexColor");
gl.enableVertexAttribArray( shaderProgram.vertexColAttrib);

shaderProgram.perspMatrixUniform = gl.getUniformLocation(shaderProgram, "uProjection");
shaderProgram.viewMatrixUniform = gl.getUniformLocation(shaderProgram, "uModelView");
```
Compiling shaders

• Create both shaders
  • gl.createShader( gl.VERTEX_SHADER )
  • gl.createShader( gl.FRAGMENT_SHADER )

• Set the source file - gl.shaderSource( shaderObj, src )

• gl.shaderCompile( shaderObj )

• After fragment and vertex shaders are compiled, we attach them to a shader program
  • gl.createProgram()
  • gl.attachShader( shaderProgram, shaderObj )

• Then we need to link the program to be able to use it.
  • gl.linkProgram( shaderProgram )
  • gl.useProgram( shaderProgram )
Transferring data

• Need to define link between data in application memory and GPU memory
  • Transferring bytes
  • Tell GPU how to read this data
• Bit tedious process, but only have to do it once
• Done through Vertex Buffer Objects (VBO)
  • Create buffer of required size ( gl.createBuffer(…) )
  • Bind it, so it is actually used ( gl.bindBuffer(…) )
  • Fill the bound buffer with data ( gl.bufferData(…), gl.bufferSubData(…) )
Transferring data

```javascript
var floatByteSize = 4;

// specify the data arrays
var positions = [0.0, 1.0, 0.0,
                 -1.0, -1.0, 0.0,
                 1.0, -1.0, 0.0];

var colors = [1.0, 0.0, 0.0,
              0.0, 1.0, 0.0,
              0.0, 0.0, 1.0];

// create buffer
var triangleData = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, triangleData);

// upload the data
gl.bufferData(gl.ARRAY_BUFFER, (colors.length * positions.length) * floatByteSize, gl.STATIC_DRAW);
gl.bufferSubData(gl.ARRAY_BUFFER, 0, new Float32Array(positions));
gl.bufferSubData(gl.ARRAY_BUFFER, positions.length * floatByteSize, new Float32Array(colors));

// Helpers variables for gl.VertexAttribPointer
triangleData.attribSize = 3;
triangleData.numVerts = 3;
triangleData.colorOffset = positions.length * floatByteSize;
return triangleData;
```
We are almost able to draw the triangle!

Exciting!

Still a couple of steps

Need to bind the buffer we are drawing

Need to explain to WebGL how to read data off the buffer

`gl.vertexAttribPointer( attribLocation, attribSize, type, normalize, stride, offset)`

- location from shader program
- number of components per vertex
- space between elements (bytes)
- starting offset (bytes)
- type of the data ( `gl.FLOAT` )

normalize or take as-is (boolean)
Drawing

- Based on the way your data is stored you can draw it by invoking
  - `void gl.drawArrays ( enum mode, int first, long count)`
  - `void gl.drawElements ( enum mode, long count, enum type, long offset )`

  - `mode : POINTS, LINE_STRIP, LINE_LOOP, LINES, TRIANGLE_STRIP, TRIANGLE_FAN, TRIANGLE`
  - `type : UNSIGNED_BYTE, UNSIGNED_SHORT`

- `gl.drawArrays(…)` just reads the values as take come from the buffer
- `gl.drawElements(…)` requires ELEMENT_ARRAY_BUFFER to be bound to specify reading order
Transformations

- We specify our data in 3D space, while the end result is 2D image
- We need to perform series of transformation
  - Model matrix - objects in 3D has its own transformation matrix
  - View matrix - camera has position and orientation
  - Projection matrix - camera’s intrinsic parameters
- These three matrices model how your data will be displayed
- JavaScript library for vector/matrix operations: [link]
  - Matrix manipulation
  - Useful constructors - perspective camera, orthographical camera
Transformations

• Understanding the transformations between each coordinate space is crucial for graphic programming

• Good tutorial on the topic: link

Image credit: http://www.opengl-tutorial.org
Adding 3D to our app

• We need to modify our buffers with 3D data.
• We can use `ELEMENT_ARRAY_BUFFER` to specify exact triangle indices.

```javascript
var indices = [
  0, 1, 2, 0, 2, 3, // bottom
  0, 1, 5, 0, 4, 5, // front
  1, 2, 6, 1, 5, 6, // left
  2, 3, 7, 2, 6, 7, // back
  3, 0, 4, 3, 4, 7, // right
  4, 5, 6, 4, 6, 7 // top
];

gl.bindBuffer( gl.ELEMENT_ARRAY_BUFFER, cube.indices);
gl.bufferData( gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);`
Animation

• In the examples we use `requestAnimFrame()`
• Does not refresh if tab is not active
• Defines rendering loop

```javascript
function tick() {
    requestAnimFrame(tick);
    drawScene(gl, shaderProgram, camMatrices, cube);
    animate(cube);
}

function animate(mesh) {
    var timeNow = new Date().getTime();
    if (lastTime !== 0) {
        var elapsed = timeNow - lastTime;
        mat4.rotateY(mesh.transformation, mesh.transformation, deg2rad(elapsed / 8));
    }
    lastTime = timeNow;
}
```
User Interaction - Arcball

- Arcball is an interaction method to translate \( \{x,y\} \) screen locations to a motion of an object
- Obtain two pairs of \( \{x,y\} \) screen coordinates. Normalize them to \([-1,1]\) range.
- Treat them as positions on hemisphere of radius 1
- Calculate \( z \) form sphere equation. Gives vectors \( P_1, P_2 \)
- Compute rotation angle \( \theta = \arccos(P_1 \cdot P_2) \)
- Compute rotation axis \( R = P_1 \times P_2 \)
- \( \) \( \) \( \) \( \text{rotAxis} \) exists in camera coordinates, need to move it to model coordinates

\[
R' = (V_{rot}M_{rot})^{-1}R
\]

- Your matrix library should be able to generate rotation matrix from \( \{\theta, R\} \)
Texture Mapping

• Process similar to buffer creation:
  • Create texture gl.createTexture(…)
  • Bind texture gl.bindTexture(…)
  • Configure texture (a lot of options)
    • gl.texImage2D(…) - explain image data
    • gl.texParameteri(…) - texture filtering options

• Texture units
  • Specify current set of active textures - gl.activeTexture(gl.TEXTUREX)
  • Need to explicitly state which we use

• Modify mesh data with per vertex texture coordinates
Texture Mapping

```javascript
function handleLoadedTexture( texture ) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, texture.image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_NEAREST);
    gl.generateMipmap(gl.TEXTURE_2D);
    gl.bindTexture(gl.TEXTURE_2D, null);
}

function initTexture(gl, mesh, filename ) {
    mesh.texture = gl.createTexture();
    mesh.texture.image = new Image();
    mesh.texture.image.onload = function () {
        handleLoadedTexture(mesh.texture);
    }
    mesh.texture.image.src = filename;
}
```

gl.activeTexture( gl.TEXTURE0 );
gl.bindTexture( gl.TEXTURE_2D, mesh.texture );
Texture Shader

- Texture is 2D image, rendered to a part of your output
- We need to sample our texture to get correct pixel values in the output image
- `sampler2D` object and `texture2D(...),` are the functions you need to use in your shader

```plaintext
precision mediump float;
varying vec3 vVertexColor;
varying vec2 vVertexTexCoord;

uniform sampler2D uSampler;

void main(void) {
    vec4 textureColor = texture2D(uSampler, vec2(vVertexTexCoord.s, vVertexTexCoord.t));
    gl_FragColor = vec4(vVertexColor.rgb * textureColor.rgb, 1.0);
}

//bind the used texture
.gl.activeTexture( gl.TEXTURE0 );
.gl.uniform1i( shaderProgram samplerUniform, 0 ); //this is the texture uit we are using
.gl.bindTexture( gl.TEXTURE_2D, mesh.texture );
```
Thanks!